統語構造アノテーション支援ツールの開発

著者	窪田 悠介
雑誌名	国立国語研究所論集
号	13
ページ	107-125
発行年	2017-07
URL	http://doi.org/10.15084/00001374

国立国語研究所論集 (NINJAL Research Papers) 13: 107–125 (2017)
ISSN: 2186-134X print/2186-1358 online

統語構造アノテーション支援ツールの開発

窪田悠介

筑波大学／国立国語研究所 共同研究員

要旨

本稿では，統語構造アノテーション支援ツール Emacs けやきモードの解説をする。けやきモードは，国立国語研究所「統語・意味解析コーパスの開発と言語研究」プロジェクトのために開発された。本ツールを開発する過程で，Emacs をテキストアノテーション作業用インターフェイス構築の土台として利用する手法の有効性と，この手法を採用する際に注意すべき点がいろいろと明らかになった。主な利点は，Emacs エディタに備わっている Emacs Lisp と呼ばれる Lisp の方言を用いることで，強力なテキストアノテーション支援環境を素早く開発できることである。同時に，当初開発者側に盲点となっていたがツールを現場で運用する際に徐々に明らかになった落とし穴として，Emacs のデフォルトのインターフェイスの使いにくさがあることが分かった。本稿では，けやきモードの主な特徴と実装を簡単に説明したあと，Emacs をアノテーション支援ツール開発の基盤として用いることの利点と落とし穴を議論する *。

キーワード：ツリーバンク，アノテーション，コーパス，Emacs，ユーザー・インターフェイス

1. はじめに

本稿では，筆者が開発している統語構造アノテーション支援ツール **Emacs けやきモード**の解説をする。けやきモードは，テキストエディタ Emacs 上に実装された構文木編集ツールであり，国立国語研究所（国語研）「統語・意味解析コーパスの開発と言語研究」プロジェクト（NINJAL Parsed Corpus of Modern Japanese [NPCMJ] プロジェクト）におけるアノテーター支援のために開発された[1]。本ツールを開発する過程で，より一般的に Emacs をテキストアノテーション作業用インターフェイス構築の土台として利用する際に参考になると思われる知見がいろいろと得られ，これらの知見は出版物の形で公開するに値すると判断したことが本稿執筆の主な動機となっている。このため，本稿では，けやきモードの具体的な特徴の説明とともに，コーパス開発におけるアノテーション支援ツールの作成に際して発生する実際的な諸問題に NPCMJ プロジェクトにおいて我々がどう対処したかもあわせて議論する[2]。

大規模なコーパス開発プロジェクトにおいては，個々のプロジェクトの内部で，様々なコーパス開発用のツールが作成されていることと思われるが，管見の限りでは，そのようなツールの作

* 本稿は国立国語研究所機関拠点型基幹研究プロジェクト「統語・意味解析コーパスの開発と言語研究」（プロジェクトリーダー：プラシャント・パルデシ）の研究成果である。けやきモードの開発，また本稿の執筆にあたっては，以下の方々から貴重なフィードバックをいただいた。ここに記して謝意を表したい（敬称略）。Prashant Pardeshi，Alastair Butler，吉本啓，Stephen Horn，窪田愛，国立国語研究所「統語・意味解析コーパスの開発と言語研究」プロジェクト・アノテーターの皆様。

[1]「けやきモード」という名称は，NPCMJ の前身が「けやきツリーバンク」と呼ばれていたことに由来する。

[2] コーパス・アノテーション全般に関する概説としては，Pustejovsky and Stubbs（2013）がよい。

成にまつわる諸問題が個別のプロジェクト内の共同体の外に出て，より広いコーパス開発・研究に関わる学問的営みの場で議論されることは稀である。今後コーパス研究が言語研究においてさらに重要性を増すと考えられること，また，コーパス開発に関わる言語学者は多くの場合プログラミングやインターフェイス設計の専門家でないため，手近に得られる情報が少ないとコーパス開発へのハードルが余計に高くなってしまうという事情を考えると，ツール開発に関する知見が，個々のプロジェクトが終了するやいなや散逸し失われてしまう，あるいは，個人的な「口コミ」によってしか知り得ない，いわば秘術のようなものとして細々と伝承されていくという状況は，学界の発展にとって好ましくないと思われる。このような状況に一石を投じたいと思ったため，本稿を執筆することにした。

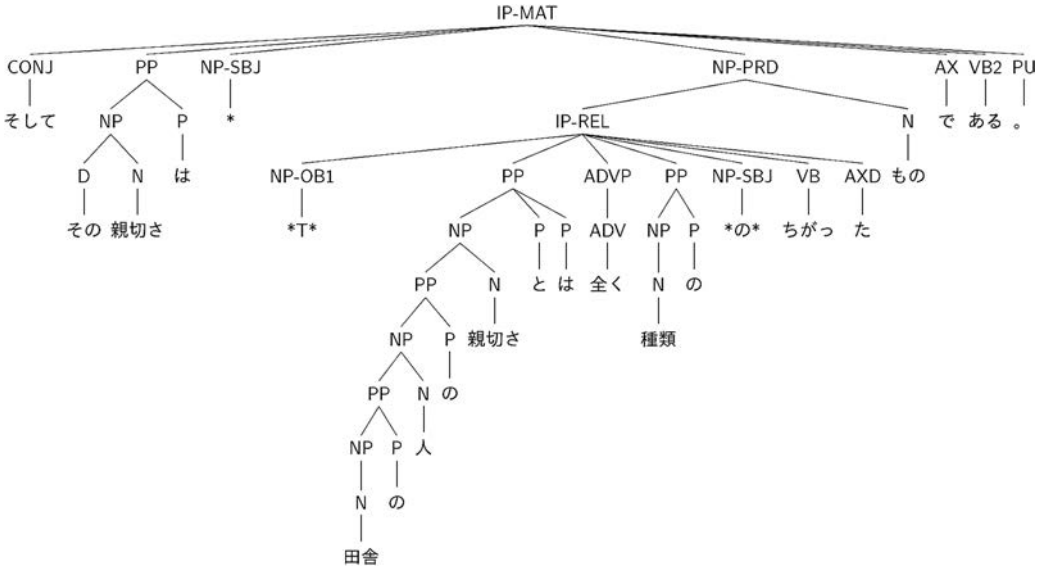
2. アノテーション支援ツール開発の経緯

2.1 背景

NPCMJ コーパス（http://npcmj.ninjal.ac.jp/）は，現在国語研において開発中の，テキストに対して統語構造を付与した，いわゆる**ツリーバンク**と呼ばれるタイプのコーパスである。アノテーション基準はペン通時コーパス（Santorini 2010）の方式を日本語に合わせて調整したものを用いており（NPCMJ プロジェクト 2016），(1) に示すようなプレーンテキスト形式で統語構造のアノテーションを行っている（アラステア＝バトラー他 2016）。(1) のツリーは，言語学でより一般的に用いられる (2) に示した樹形図での表記と，表示している情報は等価である。

(1)
```
( (IP-MAT (CONJ そして)
          (PP (NP (D その)
                  (N 親切さ))
              (P は))
          (NP-SBJ *)
          (NP-PRD (IP-REL (NP-OB1 *T*)
                          (PP (NP (PP (NP (PP (NP (N 田舎))
                                                 (P の))
                                             (N 人))
                                         (P の))
                                     (N 親切さ))
                                 (P と)
                                 (P は))
                          (ADVP (ADV 全く))
                          (PP (NP (N 種類))
                              (P の))
                          (NP-SBJ *の*)
                          (VB ちがっ)
                          (AXD た))
                  (N もの))
          (AX で)
          (VB2 ある)
          (PU 。))
  (ID 71_denen_zakkan;NJ))
```

(2)



確率付き文脈自由文法パーザーにより出力された統語解析結果を目でチェックしてエラーを直すのが，人間のアノテーターの行う作業の中心となる[3]。データはプレーンテキストなので，アノテーション作業の方法としては，主に以下の三つが考えられる。

1. テキストエディタを用いて，もともと備わっている編集機能のみで直接テキストデータを編集する
2. アノテーション作業に特化したツールを用いる[4]
3. テキストエディタにマクロを追加して，頻繁に行う編集操作をショートカット・キーやマウスを使ったメニュー操作で行えるようにする

それぞれ一長一短があるが，現在 NPCMJ プロジェクトでは，3 の方法を採用している。紙幅の都合上, 1 と 2 との詳細な比較は省くが, 1 の方法では, プレーンテキストの括弧表示によるツリー構造を直接編集してノード移動などの複雑な操作をするのはかなり煩雑な作業であるという問題がある。NPCMJ コーパスでは，ゼロ代名詞や関係節の「トレース」の情報，また等位構造の正確な統語構造など，きめ細かな統語情報を付与している。この点で，形態素解析結果か，あるいは，せいぜい表層の語句の係り受け情報のみしか付与していない，BCCWJ などの従来のコーパスと性質が大きく異なる。アノテーターは，ただでさえ言語学的に高度な判断をするという認知的に負荷の高いタスクに従事しているので，編集作業自体にかかる負担はできるだけ少ないほうが望ましい。2 の方法は，この問題を解決するが，こちらにも問題がある。この方法では，ツー

[3] コーパス構築作業全体の流れに関しては，吉本（2016）を参照。
[4] このようなツールとしては，たとえば，Annotald（https://annotald.github.io/）がある。

ル開発の時点で，あらかじめ人間のアノテーターが行う可能性のある編集作業をある程度網羅的に把握して，それに対応する機能をすべて実装しておく必要がある。NPCMJ プロジェクトにおいては，プロジェクト開始時（2016 年 4 月），日本語に句構造情報を付与した十分に大規模なコーパスは皆無という状況だった。そのため，適切な既存のツールがそもそも存在せず，またそのようなものを一から構築することにも大きな時間的･金銭的コストがかかることが予想された。ツールの開発に手間取って実際のアノテーション作業の開始が遅れたりしては本末転倒である。3 の方法は，いわば，1 と 2 の利点を組み合わせた妥協案といえる。我々の置かれた現実的な状況においては，この妥協案を採り，アノテーション作業の遂行と並行する形でツールを開発するのが最善であるという判断に至り，けやきモードの開発に着手した。

2.2 何故 Emacs か？

我々は，そこで上記 3 の手法を用いて，Emacs エディタにマクロを追加してアノテーション支援ツールを開発することにした。この方法には，以下の利点がある [5]。

1. エディタにもとから備わっているテキスト編集機能と，マクロで追加したショートカットを組み合わせることで柔軟かつ高速に編集を行うことができ，2.1 節で指摘した 1 の手法と 2 の手法の問題点をともに克服できる
2. Emacs は，エディタ自体が **Emacs Lisp** と呼ばれるプログラミング言語 Lisp の方言によって書かれており，特定の用途のための大がかりな拡張機能を追加することが容易である
3. Lisp の「**ボトムアップ・プログラミング**」の手法が，「アノテーション作業の遂行と並行してツールを開発する」というアノテーション・ツール開発の現場でしばしば生じる必要性にうまく合致している
4. Lisp は Haskell や Scala などの**関数型言語**の祖型ともいえる特徴を持つ，非常に強力な言語であり，**高階関数**を自然に扱えるので，木構造の編集などの複雑な操作を簡潔なコードで記述できる

我々が NPCMJ プロジェクトの現場で直面していた諸問題は，その多くが，統語構造のアノテーションという特定のタスクに限らず，より一般的にコーパス開発の現場でしばしば生じがちな問題であると考えられる。このため，これらの問題にうまく対応できる Emacs を用いた開発手法は，幅広い応用の可能性を持つ。この点に関しては，4.2 節でさらに議論する。

[5] これに対して，欠点としては，Emacs Lisp を読み書きできる開発者を見つけるのが難しいという点が挙げられる。我々のプロジェクトでは，たまたま Emacs のヘビー・ユーザーである筆者がプロジェクトに関わっていたため，この問題は生じなかったが，そのようなケースは比較的稀であると考えられる。この点に関しては，4.2 節で議論する。

3. けやきモードの概要

3.1 Emacs について

けやきモードは Emacs エディタの**メジャーモード**として実装されている。この点を理解し，このあとの議論をスムーズに追うためには，まず最初に Emacs のテキストエディタとしての特徴をおおまかに把握しておいたほうがよいと思われるので，けやきモード本体の説明をする前に Emacs エディタの特徴をごく簡単に説明しておく[6]。Emacs は主に UNIX 系 OS において広く使われているオープンソースのテキストエディタであり，「ライバル」である vi と比べると，テキストエディタとしては過剰とも思われるほど拡張性が高い点が大きな特徴の一つである。この拡張性の高さを支えているのが，**Emacs Lisp** と呼ばれる Lisp の方言である。

Emacs Lisp は，最も表面的なレベルにおいては，Emacs のカスタマイズ用の言語，つまり，他のテキストエディタでいえばマクロ言語に相当するものであるが，一般的なエディタのマクロ言語と違い，一つの完全なプログラミング言語といってもいいほどの機能を備えている[7]。マクロ言語が完全なプログラミング言語であるということは，要するに，ユーザーが自分で好きなようにエディタの機能を拡張することがほとんど無制限にできる，ということを意味する。この拡張性の高さを利用して，Emacs には様々な拡張機能を提供するパッケージが世界中のユーザーによって制作されている。これらのパッケージは，大抵，何らかの目的に特化したコマンド群と，それらのコマンドを呼び出すインターフェイス（独自のキー定義など）をセットにした，「メジャーモード」と呼ばれる環境定義一式として提供されている。ユーザーは自分の使いたいパッケージを Emacs にインストールし，そのパッケージが提供するモードを必要に応じてオンにしたりオフにしたりして利用する。このことにより，Emacs は，時には LaTeX による組版の編集環境として振る舞ったり，時にはメーラーとして振る舞ったりと，時と場合に応じてユーザーに対して様々な顔を見せる[8]。

幸いなことに，Emacs と Emacs Lisp に関しては日本語で書かれた書籍が充実しているので，興味を持った読者には，アノテーション用のメジャーモードを自作してみることをお薦めしたい。淺尾・李（2013）や田野村（2012）などでカバーされている程度の基本的なプログラミングの知識があれば，広瀬（1999）と山本（2000）を教科書として読んで，るびきち（2011）をレファレンスとして使い，それで分からないことがあれば Web で調べるようにすれば，コーパス・アノテーション用のメジャーモードを自作する際に必要とされる程度の知識はすべて得ることができるはずである。

3.2 機能とインターフェイス

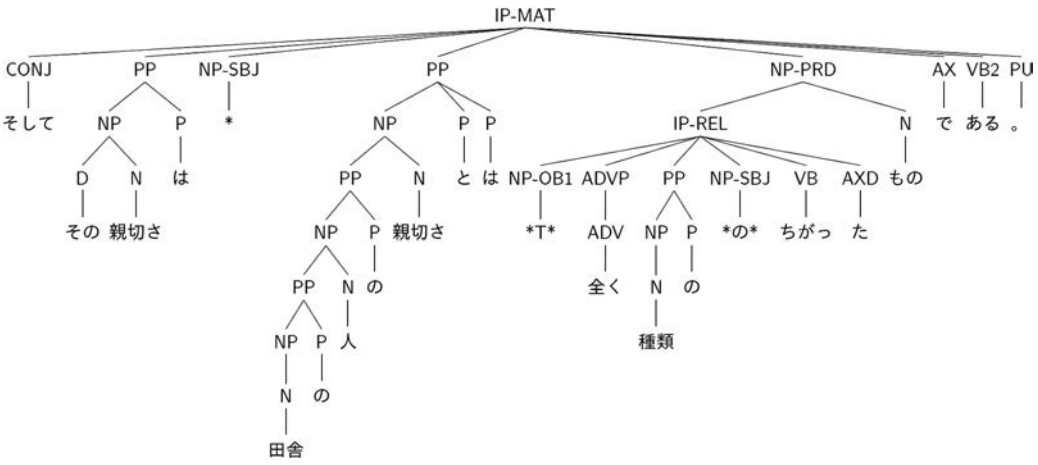
上述のように，けやきモードは Emacs のメジャーモードとして実装されており，インストールすると，2.1 節（1）に示した形式でブラケットにより統語構造をタグ付けしたテキストファイ

[6] さらに詳しくは大竹（2012）などの書籍を参照されたい。
[7] このため，Emacs 本体の基本機能の多くが，Emacs Lisp によって実装されているほどである。
[8] 仕事に疲れて気分転換したいときは，テトリスなどのゲームで遊ぶこともできる。

ルの編集時に自動的にオンになる。けやきモードの主な機能は，編集操作の補助と外部プログラムとの連繋に大別される。また，インターフェイスとしては，良く使う機能をキーボードの F1–F12 のファンクションキーに割り当て，これと，マウスを利用したプルダウンメニューや右クリックメニューによる GUI 操作とを適宜使い分ける方法を採っている。

編集補助機能のうち，最も頻繁に用いられ，また，利便性の高い機能は，ノード移動の操作である。(3) の PP ノード「田舎の人の親切さとは」のアタッチメントを主節から埋めこみ節に直す作業を例にとって，アノテーターが行う一連の操作を説明する。

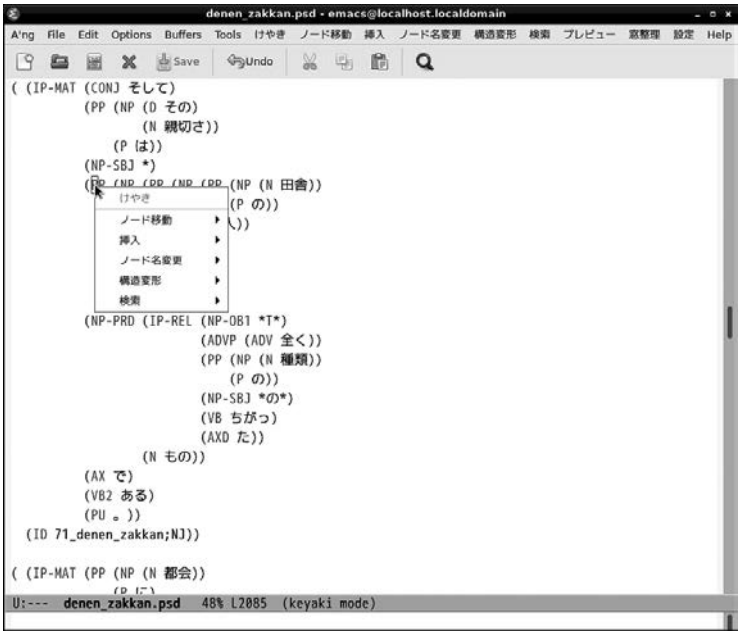
(3)



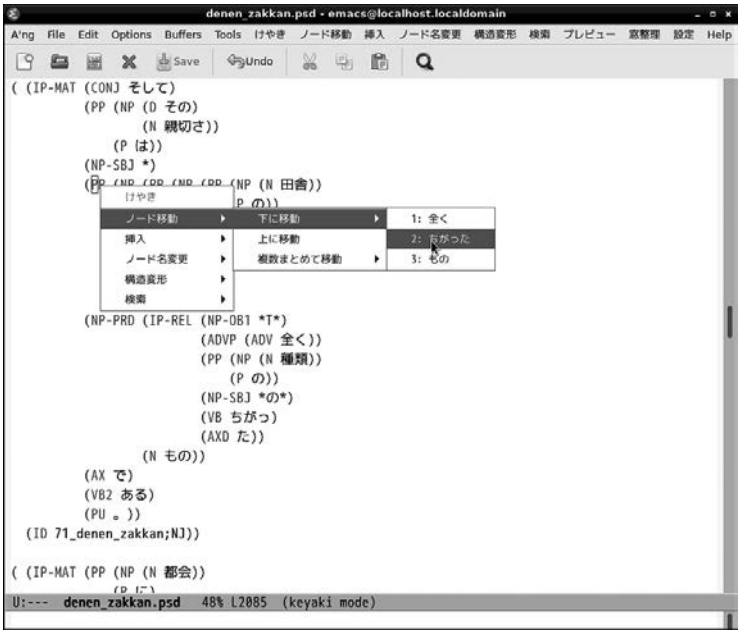
ノード移動の操作は，以下のような流れで行う。

・移動対象のノードを右クリックすると，(4) のようにメニューがポップアップする。
・アノテーターは，選択肢から該当する操作を選ぶ。現在の例の場合，アタッチメント位置を下げるので，「ノード移動－下に移動」を選択する。
・すると移動先の候補が (5) のように移動先の構成素の末尾文字列として表示されるので，該当する移動先を選ぶ。現在の例では，2 番目の候補が正しいので，それを選択する。
・すると，ノードの移動が行われ，テキストが (1) の形に書き変わり，プレビューアーの表示も自動で (2) のツリーに更新される。

(4)


denen_zakkan.psd - emacs@localhost.localdomain
A'ng File Edit Options Buffers Tools けやき ノード移動 挿入 ノード名変更 構造変形 検索 プレビュー 窓整理 設定 Help
Save Undo
((IP-MAT (CONJ そして)
(PP (NP (D その)
(N 親切さ))
(P は))
(NP-SBJ *)
(NP (N 田舎))
(P の))
けやき
ノード移動
挿入
ノード名変更
構造変形
検索
(NP-PRD (IP-REL (NP-OB1 *T*)
(ADVP (ADV 全く))
(PP (NP (N 種類))
(P の))
(NP-SBJ *の*)
(VB ちがっ)
(AXD た))
(N もの))
(AX で)
(VB2 ある)
(PU 。))
(ID 71_denen_zakkan;NJ))
((IP-MAT (PP (NP (N 都会))
U:--- denen_zakkan.psd 48% L2085 (keyaki mode)

(5)

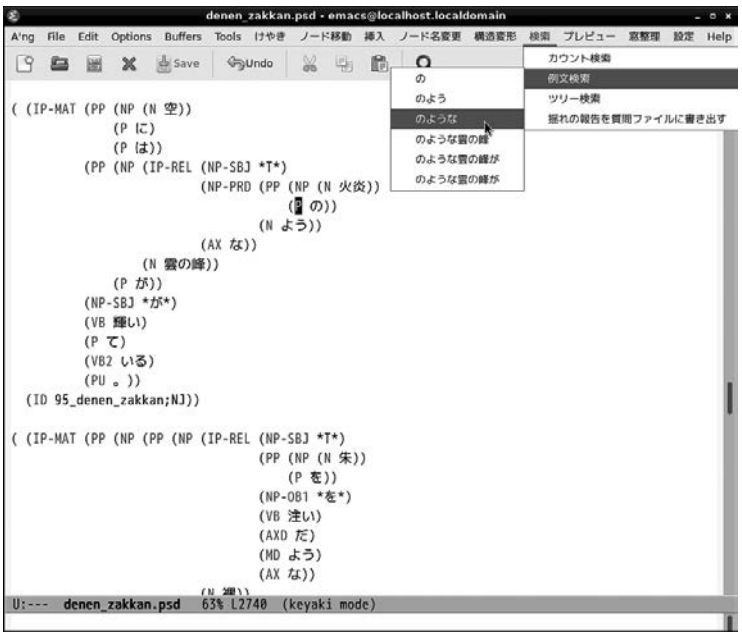

denen_zakkan.psd - emacs@localhost.localdomain
A'ng File Edit Options Buffers Tools けやき ノード移動 挿入 ノード名変更 構造変形 検索 プレビュー 窓整理 設定 Help
Save Undo
((IP-MAT (CONJ そして)
(PP (NP (D その)
(N 親切さ))
(P は))
(NP-SBJ *)
(NP (N 田舎))
けやき
ノード移動
挿入
ノード名変更
構造変形
検索
下に移動
上に移動
複数まとめて移動
1: 全く
2: ちがった
3: もの
(NP-PRD (IP-REL (NP-OB1 *T*)
(ADVP (ADV 全く))
(PP (NP (N 種類))
(P の))
(NP-SBJ *の*)
(VB ちがっ)
(AXD た))
(N もの))
(AX で)
(VB2 ある)
(PU 。))
(ID 71_denen_zakkan;NJ))
((IP-MAT (PP (NP (N 都会))
U:--- denen_zakkan.psd 48% L2085 (keyaki mode)

日本語は主要部末尾言語であるため，移動先の候補が末尾文字列で表示されるようにしてある[9]。アノテーターは，ブラケット表示をどのように編集するかといった，エラー修正の本質とは関係のない煩雑なエディタ編集操作の詳細に煩わされることなく，「当該の構成素をどの述語にかかる要素として再分析すべきか」という言語学的な問題に集中することで必要な操作を完了することができる。これにより，編集操作が容易になるだけでなく，テキストを手で直接編集することにより括弧の対応関係を崩してしまうといったエラーも防ぐことができる。

編集補助機能には，ノード移動以外に，(i) 空要素や文法関係タグの挿入，(ii) 品詞タグの修正，(iii) ノード移動以外の各種構造変形の機能がある。

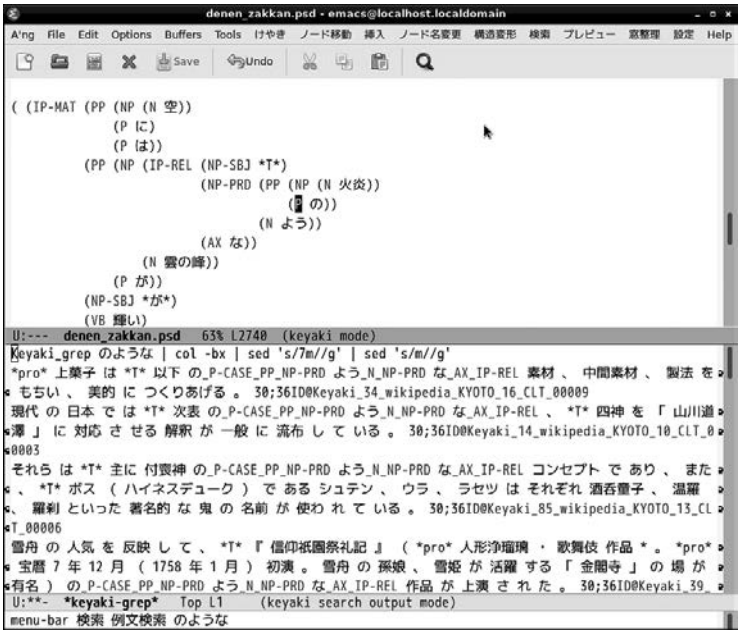
編集補助機能以外に，けやきモードには外部プログラムとの連繋機能も備わっている。Emacs にはもともと外部プログラムとの連繋を取る機能が備わっているため，これを用いて，プレビューアーへの編集結果の即時反映や，編集中の文に表れる文字列に対するタグ付けを修正が完了しているファイルのデータベースから検索する機能などを実装している。(6)（7）に検索機能の使用の画面例を示す。この例では，文中に表れた「のような」という文字列に対するタグ付けを修正済みデータベースから検索している。

(6)



[9] 英語などの他言語用に流用する際は，表示する情報とその表示方法を調整する必要があると考えられる。

(7)



NPCMJ プロジェクトで構築しているツリーバンクのようなコーパスにおいては，言語学的にかなり詳細な情報をタグ付けするため，アノテーション基準が非常に複雑かつ複合的になる。このため，アノテーターは，修正作業における個々の事例に対して，アノテーション・マニュアル，修正済みファイル内の類似例，日本語の文法に関する記述的一般化，さらには一般的な言語学的知識までを総合的に参照して最終的な判断を下す必要に絶えず迫られている。複数のツールを PC で同時に立ち上げて，さらに紙媒体の辞書や文法書なども必要に応じて参照する，というような形で作業を行うと，トレーニング中はともかく，スピードを要求される実際のアノテーション作業においては非常に効率が悪い。このため，編集作業を行っているツール上から必要な情報を素早く参照できるようなインターフェイスを整えておくことは，効率的なアノテーション作業にとって必須の要件である。けやきモードの外部プログラムとの連繋機能は，この要請を満たすことを目的として作られている。

3.3 コードの実装

けやきモードのコードは，以下の URL において公開している。

http://www.u.tsukuba.ac.jp/~kubota.yusuke.fn/emacs-keyaki-mode.html

以下では，プログラムの中核部分がどのように実装されているかを簡単に説明する。Emacs Lisp においては，テキスト文書の内容を編集する関数は，すべて，編集対象の文字列の文書内でのポイント位置を参照して編集操作（文字列の削除，追加，書き換えなど）を行う。このため，

木構造編集のプログラムを書く際は，ツリー内のノードを文書内でのポイント位置で直接参照するのが最も手っとり早い。そして，「ノードを別の場所に移動する」というような複雑な操作を行う関数を書くためには，その「部品」として，操作対象のノードの位置情報から親ノードや兄弟ノードの位置を得る関数を用意しておく必要がある。

これらの小さな「部品」のうち最低限の基本的なもの（たとえば，子ノードのポイント位置を与えると親ノードのポイント位置を返す関数）は，括弧構造を直接参照する，いわば「低レベル」の関数として書く必要がある。Lisp は，以下の（8），（9）に示したコード例からも分かるように，もともとプログラム自体が括弧構造で書かれている。Lisp と親和性の高いエディタである Emacs Lisp には，このような括弧構造を操作する基本的な機能がすでに備わっているため，それを用いれば，これらの「低レベル」関数を書くことは容易である [10]。

さて，これらの「低レベル」の関数群を利用して，より複雑なノード間の関係を扱う関数を定義することができる。たとえば，あるノードを含む構成素内の先頭のノードに何らかの操作をしたいといった場合を考えてみよう [11]。こういった操作をする際には，まず現在ノードの兄弟ノードのうち一番先頭に現れるものを同定する関数が必要となるが，このような関数は，テキスト内の括弧位置などに直接言及する形で定義するよりも，すでに定義してある「低レベル」の関数群を組み合わせて定義するほうが簡単である。（8）に定義を示す。

(8)
```
(defun get-first-sister (node)
  (get-first-daughter (get-mother node)))
```

この関数定義は，**node** の「一番先頭の兄弟」を得るためには，**get-mother** で親を取って，さらに **get-first-daughter** を使ってその最初の子を取れば良い，ということを意味している [12]。ここで，**get-mother** と **get-first-daughter** は，「低レベル」の関数としてあらかじめ定義済みのものである。

実際のコーディングでは，このような関数を必要なだけ定義して，それらを用いて，ノードの移動や追加などといった実質的な操作をする，より複雑な関数を書いていくことになる。ここで重要なのは，木構造におけるノード間の関係性を規定する関数と，木構造が実際のテキストでどう表現されているかを規定する関数とを別の「部品」として独立させておくことで，プログラムの大部分を「木構造が具体的にどういう形で表現されているか」ということと切り離して，純粋にノード間の兄弟関係や支配関係のみに言及することで書くことができることである。このことによって，関数の定義が読みやすくなるという保守性におけるメリットが得られるだけでなく，木構造の表現形式が変更された場合でも，「低レベル」の処理を行う部分を書き換えるだけでプ

[10] 本稿の目的は Emacs Lisp プログラミングの技術的な詳細を議論することではないので，これらの関数の説明はここでは省略する。興味がある方は，適宜，広瀬（1999）や，るびきち（2011）などを参照しつつ，直接コードを読んでいただきたい。
[11] けやきモードでは，文の述語にカーソルを合わせてその述語の項であるゼロ代名詞を文頭に挿入する機能などをこの種の操作で実現している。
[12] Haskell などのより高度な関数型言語では，関数合成を用いて，**get-first-sister** を **get-mother** と **get-first-daughter** の合成関数としてより簡潔に定義できるが，Emacs Lisp では残念ながら関数合成は言語の機能としては備わっていない。

ログラムのより複雑な部分は全く変更なしでそのまま流用できる，という拡張性におけるメリットも得られる。

さらに，Lisp には再帰関数が簡潔に書けるという利点があり，これが木構造のような再帰的データ構造の編集に大変役立つ。再帰関数によって，たとえば，ある条件を満たすノードすべてに対して何らかの操作をする，というような強力な編集操作を行う関数を簡単に書くことができる[13]。ここでは，簡単な例を一つだけ挙げて再帰的関数定義の概略を説明しておく。

(9)
```
(defun get-ancestors (node list)
  (if (root-p node)
      list
    (cons (get-mother node) (get-ancestors (get-mother node) list))))
```

この関数は，ノードを引数に取り，そのノードの先祖すべてからなるリストを返す。第二引数である **list** は，再帰でノードをたどっていくたびにそれまでに遭遇した先祖をすべて格納しておく役割を果たす。再帰関数は，基底部と繰り返しの定義の二つの部分からなる。この関数の基底部では，ノードがツリー全体の根ノードである（テスト **root-p** が真となる）場合，それ以上先祖はないので，そこまでで遭遇した先祖をすべて格納してある **list** をそのまま返す。繰り返し部分（4 行目）は，親の先祖に親を加えた（**cons** した）ものが自分の先祖である，ということをいっている。先祖関数を定義するために先祖関数そのものを使っているため，この関数は再帰関数である。この関数の定義が，定義の中で定義している関数自体を呼んでいるにもかかわらず循環的な定義にならず正しく意図した動作をするのは，入れ子になっている関数呼び出しを順にたどっていけば，いつかは根ノードにたどり着いて基底部の条件を満たし，繰り返しが終わるからである。

4. 課題と応用の可能性

けやきモードを作成し，それを実際にツールとして NPCMJ プロジェクトの現場で利用することを通して得られた知見のうち，ある程度一般性があると考えられる点をここでいくつか議論する。類似のプロジェクトで似たような状況が生じた際の参考になれば幸いである。

4.1 インターフェイスの設計における課題とそれに対する対応

インターフェイスの設計においては，開発者側からはユーザーが実際にどのような行動を取るかが分かりづらいということが常に問題となる[14]。けやきモードの開発においても，この点で当初一つのつまづきがあった。

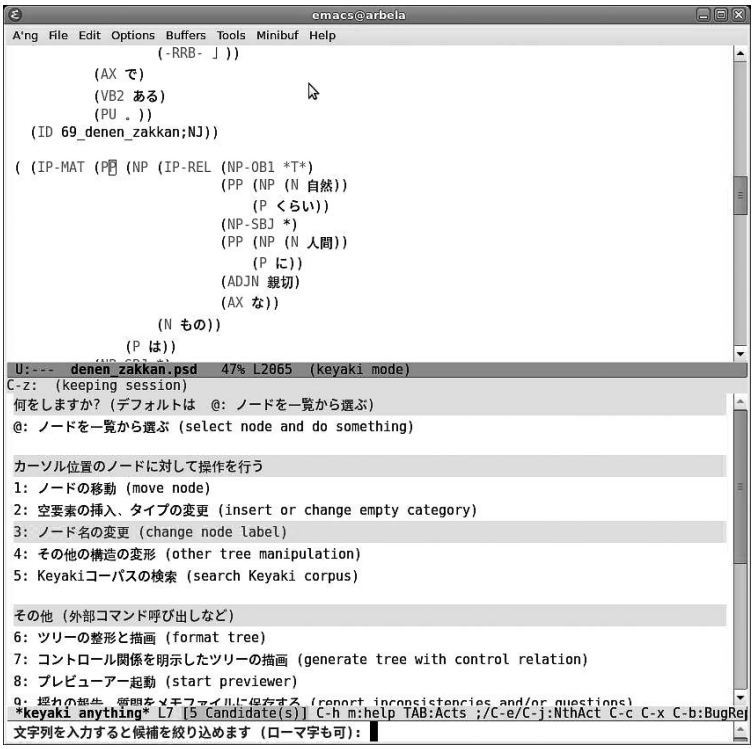
---

[13] これが役に立つのは以下のような場合である。いわゆる繰り上げ（raising）構文やコントロール構文における主節と埋めこみ節における項の同定が正しく行われるかということに関しては，アノテーション作業において特別な注意を払う必要があるため，けやきモードでは，主語が明示されていない節をプレビューアーでカラーコーディング（具体的には，赤字で表示）することでアノテーターの注意を促している。これを実装するには，ツリーを根から順にたどって，「節であり，かつ直下に主語ノードがない」という条件を満たすノードにだけ印をつけていけばよいわけだが，このような操作には再帰関数が非常に有用となる。

[14] 中村（2015）は，現実の世界において発生してしまった「バッドユーアイ（=Bad User Interface）」の実例を多数挙げながら，この点を非常に分かりやすく議論しており，参考になる。

3.1 節で述べたように，現在のけやきモードでは，マウスを使って操作する GUI インターフェイスを基本としている。最初のバージョンではこの機能は実装しておらず，各種操作選択のためのメニュー・インターフェイスを,「Anything モード」と呼ばれるキーボードベースの Emacs のインターフェイス拡張機能を用いて実装していた。(10)に, 当初のインターフェイスの画面例を示す。

(10)



けやきモードの開発者である筆者は，Emacs を長年使っており，Emacs では基本的にキーボードのみですべての編集作業を行うものであるという固定観念に囚われていたため，①キーボードのみのインターフェイスが, 果たして Emacs に馴染みのないアノテーターにとって使いやすいか, また，②そもそも（テキスト編集一般ではなく）アノテーション作業用のツールとして考えた場合に, キーボードのみで操作するという方式がインターフェイスとして最適のデザインであるか, という二点に検討の余地があることを当初見落としていた。

実際にけやきモードを作って国語研のアノテーターに利用してみてもらったところ，プレビューアーとの連繋などの機能はほぼすべてのユーザーが使っているが，ノードの移動などの操作は手で直接編集することで行っている人も多い，ということが判明した[15]。特に問題点として

[15] 実際にアノテーターが作業しているところを観察させてもらい意見交換をしたり，使いにくい点をアンケート調査や，メールでのやりとりで報告してもらうなどの方法で，利用状況・使い勝手の調査を行った。

浮かび上がったのが，Emacs のインターフェイスの古色蒼然さと奇妙さである[16]。

現在の Windows や Mac の一般的なワープロソフトなどだと，たとえばファイルを開くといった操作をする際は，編集画面とは別のダイアログボックスが立ち上がるようになっている。ところが，Emacs では（少なくとも，キーボードで操作をする場合），（10）に示した画面と同じように，まず画面自体が二つに割れ，下の画面のほうに開くファイルの候補リストが表示される。また，ユーザーによる候補選択も，画面の一番下の「ミニバッファ」と呼ばれる行にキーボードから文字を入力することによって行うのが基本である。

この「編集対象のテキストが入っていた画面がいきなり二つに割れる」という点と，「候補選択モードではカーソルが画面下のミニバッファに移る」という点は，Emacs に慣れていないユーザーにとっては非常に違和感のあるインターフェイスである[17]。計算機の速度が遅かった時代には GUI のダイアログボックスの実装を省くことはリソースの節約という利点があったかもしれないが，現代的な観点からは，これは完全に時代後れな設計である。

また，アノテーション作業においては，作業の性質上，ほとんどのユーザーが，Emacs の画面と，プレビューアーなどの他のプログラムの画面とを随時行ったり来たりしながら作業を行っている。このため，① Emacs 作業中にショートカットを起動して候補選択画面を出し，②そのあとプレビューアーを念のため確認するなどのためにいったん Emacs を離れ，③再度 Emacs に戻り編集作業を再開しようとする，というようなユーザーにとっては至極自然な一連の操作の流れが何故かうまくいかずに慌てる，というような事態が頻発していた。この原因は③の時点で①で呼び出した候補選択画面がまだアクティブなので，それを解除しないと直接ファイルを編集することができないからなのだが，一度 Emacs の画面を離れたらそんなことは忘れているし，上で述べたように，Emacs のデフォルトのインターフェイスでは候補選択の状態とテキスト編集の状態の区別が非常に分かりにくいので，Emacs に不慣れなユーザーは，何が起こったのか全く分からなくなってしまう。アノテーターの中には，この点の煩わしさが原因で，けやきモードの候補選択機能の使用をそもそもやめてしまう人もいたようである。

3.1 節で紹介したプルダウンメニューとマウスの右クリックメニューは，この問題に対処するために後から付け加えたインターフェイスである。これらの機能は，マウスを使って操作するため，Emacs に馴染みのないユーザーでも直感的に利用することができる。また，インターフェイスの操作中はメニューのパネルが編集テキストの画面とは別に表示されるので，ユーザーに対し

16 現代的なインターフェイスに慣れたユーザーにとってこれがいかに使いづらいものであるかについては，松山（2009）の指摘が的を射ており分かりやすい。なお，蛇足かとも思われるが，念のため，ここで Anything モードとその後継である helm モードの擁護をしておきたい。Anything モードは，Emacs の利用にある程度慣れており，キーボード主体の操作に違和感のないユーザーにとっては，Emacs のもともとのインターフェイスの不完全さを補ってくれる素晴らしいパッケージである。筆者自身，常用してその恩恵にあずかっている。

17 前者は，見た目が同じ「編集ウィンドウ」という構造物が状況に応じて二つの全く異なった役割を担うために，形式と機能の間に明確な一対一の対応付けがない点がユーザーの混乱を招く。また，後者は，「カーソルが移動して，さらに，テキスト編集の状況から，候補選択の状況に切り替わっている」というふうに状況が大きく変化しているにもかかわらず，デフォルトの設定ではそのような事態の変化をユーザーに知らせる視覚的（あるいはそれ以外の）フィードバックが非常に乏しい点に問題がある。いずれも中村（2015）の指摘する「バッドユーアイ」の基本的要件にぴたりと合致してしまっている。

てよりはっきりと「候補選択の操作中である」という視覚的フィードバックを与えるデザインとなっており，誤操作が生じにくくなっている。

上述の候補選択メニュー以外にも，まだいろいろと使い勝手を改善する余地が残っているかと思われる。インターフェイスの設計と保守・改善においては，開発者側とユーザー側とでうまく意思疎通を行うことが肝心だが，これを効果的に行うことは意外なほどに難しい。単に開発者側から「使いにくい点があれば報告してください」とユーザーにいっておくだけでは，開発者側にとって有益なフィードバックが得られることは稀である。これは，開発者とユーザーの間には知識の差があることによる。このため，極端な場合，ユーザーは，そもそもどういう改善の可能性があるかの見当すらつかない，と感じていることさえありうる。開発者側としては，ユーザーが「この辺が何となく使いにくいんだけど，どうにかならないだろうか」というふうに，はっきりと言葉になるかならないかのレベルで感じている不満や要望をうまくユーザーから引き出して，それを機能の改善に活かす工夫が必要である。このようなことは，言うは易く，行うは難しではあるが，現在のところ，我々のプロジェクトでは，開発者も時々アノテーション作業を行うことで実際にユーザーの立場に身を置くこと，また，なるべく頻繁にアノテーターと直接会って話をすることによって，開発者とユーザーとの距離を縮める努力をしている。

4.2 応用の可能性

本節では，コーパス開発ツールを Emacs 上に実装することの利点と問題点を議論する。筆者の現時点での感触としては, 扱うデータがプレーンテキストである場合（これは, xml や html など，プレーンテキストでマークアップしてある形式も含む），多くの場合，利点が問題点を上回るのではないかと思われる。

まず，Emacs Lisp で開発することで，Emacs にあらかじめ備わっているテキスト編集機能と，エディタとしてのインターフェイス（プルダウンメニューやキーボード・ショートカットなど）をそのまま利用できるという大きな利点がある。コーパス開発の現場では，しばしば，ツール開発にそれほど時間的，また人的資源を費やすことができないにもかかわらず，採用しているアノテーション基準に依存した特殊な機能の実装が必要なために既存のツールを流用することが難しい，というジレンマが生じることが予想される。Emacs で開発するという選択肢は，この問題に対する一つの有力な解決策を提示する手法として期待できる。

次に，(Emacs) Lisp でのプログラミングで推奨されている開発手法である**ボトムアップ・プログラミング**も，コーパス開発の現場において非常に有効に働いたことを指摘しておきたい。4.1 節で議論したように，このような状況で実際に便利なツールを作るためには，ユーザーの行動を観察し，そこからフィードバックを得て設計の改善に活かすサイクルが不可欠になる。言い換えると，仕様が確定していない状況で実装を始め，とりあえず動くツールを作って，それを現場からのフィードバックにより随時拡張・改良していくという「小回りの利く」開発手法を採る必要がある。このような開発手法を「ボトムアップ・プログラミング」と呼ぶ（Graham 1997）。Emacs Lisp は，Emacs 上で小さな関数（＝プログラムの部品）を作り，それをその場ですぐテス

トして，他の部品と組み合わせて複雑なプログラムに組み上げていく，という方法でプログラムを書くのが普通であり，ボトムアップ・プログラミングの手法と非常に親和性が高い。

実際，けやきモードの開発においても，ノード移動などの主要な機能の粗削りな原形をまず作成し，それを実際に使用してみてさらに作り込み，またそれ以外の付随的な機能を徐々に付け足していくという方法でツールを作成した。Emacs Lisp を用いた開発では，実際にアノテーターが利用するインターフェイスと，プログラムのコード編集の画面の両方を，Emacs の中で同時に立ち上げて，「一つの関数を書いたらそれを即座にテストする」という形で開発を行うことができるため，バグへの対応や新しい機能の追加を比較的迅速に行うことができた。

最後に，これは一般的なプログラミング作法に関わることではあるが，Lisp の関数型言語的側面を活用してコーディングしたので（Lisp で楽に書こうとすると自然とそうなる），おのずと，

A. 木構造の関係性のみに言及する抽象的な関数
B. A の関数を利用してテキストの編集を行う関数
C. Emacs 利用中に B の関数を呼び出して実行する，ユーザー・インターフェイスに関わる関数

の三つをはっきりと分ける設計方針となった。このことにより，それぞれ役割がはっきりと異なるプログラムのサブセットを必要に応じて他の用途に流用することが簡単になる。具体的には，

1. A だけを取り出して，木構造データを扱うための一般的なツール群として使う
2. B を書き換えることで，木構造がどのような方法で表示されているかによらず（たとえば xml 形式などであってもよい），A と C を使い回す
3. C を入れ換えてユーザー・インターフェイスの部分だけ換装する

といったことができるようになる。

これらの点のうち，3 は，いうまでもなく，4.1 節で議論した，初期の課題への対応において実際に役立った。また，1 に関しても，現在準備中である筆者の別のプロジェクト（NPCMJ ツリーバンクのハイブリッド範疇文法（窪田 2015, Kubota and Levine 2015）ツリーバンクへの半自動変換）において，実験的に採用している [18]。2 に関しては，まだ実際に試してみる機会を得ていないが，

[18] このプロジェクトでは Emacs Lisp を使ってテキスト変換のバッチ処理を行うということを試してみた。Emacs Lisp をテキスト処理のためのスクリプト言語として使える，というのは面白い発見だった。この点に関しては，松山（2010）が参考になる。計算機の速度が十分に速くなった現在においては，大規模なテキストマイニングでもするのでない限り，このような用途にも Emacs Lisp は十分に実用性がある。

実際，エディタのマクロ言語とスクリプト言語の間に境目がないというのは，大きなメリットとなりうる。テキスト処理のタスクにおいては，（たとえば句読点記号の一括置換などの）簡単な操作はエディタのマクロ言語を使い，（段落区切りの体裁をまとめて変えるとか，html 形式と LaTeX 形式など，異なる文書形式の相互変換をするといった）より込み入った操作にはスクリプト言語を用いる，といった使い分けが一般的である（たとえば，荻野・田野村（2011），田野村（2012）など参照）。Emacs Lisp は，このどちらの用途にも使えるので，Emacs Lisp ですべてをこなせば，複数の言語を覚える必要がなくなるし，異なるタスク間でのスクリプトやコードの流用も自在になる。このアプローチへの最大の懸念は，Emacs Lisp はそのままだと正規表現の扱いが非常に煩雑であるという点だが，この問題も，るびきち（2011）などで解説されている便利なパッケージを導入すればかなりの程度克服できる。

問題なく可能なはずである。

このように，Emacs Lisp を用いた開発手法は，より広くコーパス構築一般におけるアノテーション支援ツール開発の場面においても有力な手法となると考えられるが，Emacs Lisp がマイナーな言語であるという，脚注 5 で述べた一見致命的な弱点がある。以下ではこの点に関して少し補足しておきたい。結論を先に述べると，筆者は，現在のプログラム開発をとりまく状況においては，これはさほど深刻な問題ではないと考えている。

Lisp は，古い歴史を持つものの，プログラミング言語の世界では「括弧ばかりの謎の言語」として敬遠され，もっぱらアカデミックなコンテクストでのみ議論の対象となる晦渋な言語として，実用的なツール開発の場面からは遠ざけられてきた傾向がある。アノテーション支援ツールの開発という場面においても，そのような偏見，あるいは単に Emacs Lisp の普及度の低さのせいだろうか，Emacs Lisp を用いたインターフェイスというのは，少なくとも筆者の知る限りは存在しないようである。実際，Emacs Lisp でプログラミングを行いメジャーモードを作成するための最良の教科書である広瀬（1999）は，以下のような読者への語りかけから始まる [19]。

> 「なんで .emacs は Lisp なんだよおお…」
> と感じたことはありませんか。せめて C 言語ライクならもっとやりたいことが書けそうな気がするのに…。
>
> かくいう筆者もそうでした。学部生の頃，Lisp の授業のあとに残った印象は，二度と触りたくないという否定的なものばかりでした。最初に .emacs を見たとき，それが Lisp で書かれていることを恨みました。それでも，もともとエディタのカスタマイズをすることが好きだったのでいやいやながらたくさんの括弧と格闘し続けました。

だが，広瀬（1999）の初版が出版されたほぼ 20 年前と現在とでは，主流のプログラミング言語の仕様やプログラミング・スタイルは大きく様変わりしている。そのせいで，Lisp が一風変わった言語であるとは必ずしも言い切れない状況が最近生じつつあるように思われる。簡単にいうと，計算機の高速化や，プログラミング言語の進化により，過去 20 年ほどの間に，開発の現場で主に使われる「主流」の言語が，Lisp ではとうの昔から実装されていた機能をどんどん取り入れる形で発展するという経緯をたどった [20]。（理論）言語学者にとって最も馴染みのある例を一つだけ挙げるならば，Java や C 言語，また，ちょっとしたテキスト処理などに用いるスクリプト言語（昔は awk や Perl など，最近では Python や Ruby など）において，ラムダ式で無名関数を書き，それを高階関数（ないし，実質的に高階関数に相当する処理）に渡す，などといった

[19] .emacs というのは，Emacs のカスタマイズ用のファイルのことである。Emacs においては，カスタマイズ用の個人設定ファイルさえ，Emacs Lisp のコードとして記述するという設計になっている。
[20] たとえば，まつもと（2008）などを参照。このような主張は，多くの場合，Lisp 擁護者や Lisp 愛好者の陣営によってなされるものであり，やや偏った視点からの事態の把握ではあるが，一方で，状況のある側面を端的に捉えていることは否定しがたいと思われる。

機能は 90 年代には存在しなかった[21]。ところが，現在では，ラムダ式を書けない言語のほうがむしろ少数派である。青木（2006）や Lipovača（2012）など，Haskell などの関数型言語の教科書で明快に説明されているように，関数を引数に取る高階関数を用いてプログラムを書くという関数型プログラミングの手法を用いることによって，プログラムの記述が簡潔になり，より柔軟かつ高機能なプログラムを簡単に書けるようになる[22]。主流のプログラミング言語にこのような機能が標準装備されるようになったため，今後，関数型プログラミングの手法は実際の開発の現場でも急速に普及していくはずである。

要するに，（やや乱暴ないい方をすれば）Lisp は今や「変な言語」でもなんでもないのである。実際，3.2 節で概略を示したように，けやきモードのコードの中核は，その大部分が，言語学者に馴染みの深い木構造の概念（兄弟（sister）関係や支配（dominance）関係など）のみを参照する関数によって書かれているため，統語論や形式意味論のトレーニングを通して形式的な定義を読み書きできる言語学者であれば[23]，ごくわずかなプログラミング固有の基礎事項（デバッグの仕方や，良いコーディング・スタイルなど）を学ぶだけで，ほぼそのままコードを読み下したり，自分で拡張機能を書いて付け足したりできるようになるはずである。

5. おわりに

本稿では，NPCMJ プロジェクトで使われている統語構造アノテーション支援ツール Emacs けやきモードの概要を説明し，さらに，このツールを開発する過程で得られた，より一般的なアノテーション支援ツール開発に関わる知見についても議論した。NPCMJ プロジェクトは，主に統語論や意味論の言語理論研究におけるコーパス利用の有用性を言語学のコミュニティーに広めることを目的としている。これ自体，多くの言語学者にとっては（特に理論研究のコミュニティーにおいては），馴染みのないよく分からないもの，というのが第一印象ではないかと思われる。我々 NPCMJ プロジェクトのメンバーは，これを「食わず嫌い」の側面が強いと考えている。プロジェクトの本体においては，この「食わず嫌い」を乗り越えて，コーパスを便利なツールとして活用してもらえるように，今，皆でいろいろと「料理の仕方」を工夫している。筆者は，このようなコーパス利用の側面だけでなく，コーパス開発の側面においても，言語学者が食わず嫌いを乗り越えて次々に新しい領域を開拓していく未来を夢想している。そのためのささやかな第一歩として，言語学者である自分が，アノテーション支援ツールを実際に作ってみるという試みを

[21] Lisp にはもとからこの機能が備わっている。余談だが，筆者は，形式意味論を学んでいた学部生時代（2000 年頃）に Lisp を用いた自然言語の構文解析入門の授業を履修してこの事実を知り，軽い衝撃を受けた。当時はまだ，世間一般ではコンピューター・プログラムというのはもっぱら手続き的に書くものだと考えられていた時代だったように思う。

[22] このように書くと，一見難しいことのように聞こえるかもしれないが，実際にはそれほど難しいことではない。たとえば，田野村（2012）は近年出色のテキスト処理入門書だが，「Ruby の基礎は抽象的で難解であり」，自身「基礎を理解しないで Ruby を愛用し，日々恩恵をこうむっている」と潔く明言しながらも，実のところ Ruby の関数言語的な側面をうまく活用した分かりやすいコーディング・スタイルで，実用的なプログラムの例を多数手際よく解説しており，大変参考になる。

[23] たとえば，吉本・中村（2016）の第 2 章で扱われている程度の，集合論や関数に関するごく初歩的な事項をきちんと理解していれば十分である。

行ってみた。この経験を元に，言語学研究者の間に「プログラミングって難しそうだと思っていたけど，ひょっとしてそんなに遠い世界のものでもないのかも」というような気づきが生じることを促していきたいと考えている。

参照文献

青木峰郎（2006）『ふつうの Haskell プログラミング：ふつうのプログラマのための関数型言語入門』東京：ソフトバンククリエイティブ.

淺尾仁彦・李在鎬（2013）『言語研究のためのプログラミング入門：Python を活用したテキスト処理』東京：開拓社.

アラステア＝バトラー・吉本啓・岸本秀樹・プラシャント＝パルデシ（2016）「統語・意味解析情報付き日本語コーパスのアノテーション」『言語処理学会第 22 回年次大会 発表論文集』589–592.

Graham, Paul（1997）野田開（訳）『On Lisp』東京：オーム社.

広瀬雄二（1999）『やさしい Emacs-Lisp 講座』東京：カットシステム.

窪田悠介（2015）「言語理論研究における「ツール」としての範疇文法」『日本言語学会第 151 回大会予稿集』324–329.

Kubota, Yusuke and Robert Levine (2015) Against ellipsis: Arguments for the direct licensing of 'non-canonical' coordinations. *Linguistics and Philosophy* 38(6): 521–576.

Lipovača, Miran（2012）田中英行・村主崇行（訳）『すごい Haskell たのしく学ぼう！』東京：オーム社.

まつもとひろゆき（2008）「まつもとひろゆきのハッカーズライフ 第 11 回 Let's Talk Lisp」http://www.itmedia.co.jp/enterprise/articles/0801/29/news010.html.（2017 年 1 月 26 日確認）

松山智大（2009）「Emacs のトラノマキ 連載第 9 回 auto-complete を使おう」http://dev.ariel-networks.com/wp/documents/aritcles/emacs/part9.（2017 年 1 月 26 日確認）

松山朋洋（2010）「スクリプト言語としての Emacs Lisp」http://cx4a.org/pub/emacs-lisp-for-scripting.ja.html.（2017 年 1 月 26 日確認）

中村聡史（2015）『失敗から学ぶユーザインターフェース』東京：技術評論社.

NPCMJ プロジェクト（2016）「Keyaki Treebank/NINJAL Parsed Corpus of Modern Japanese（NPCMJ）アノテーションマニュアル」東京：国立国語研究所.

荻野綱男・田野村忠温（2011）『講座 IT と日本語 3：アプリケーションソフトの応用』東京：明治書院.

大竹智也（2012）『Emacs 実践入門』東京：技術評論社.

Pustejovsky, James and Amber Stubbs (2013) *Natural language annotation*. CA: O'Reilly.

るびきち（2011）『Emacs LISP テクニックバイブル』東京：技術評論社.

Santorini, Beatrice (2010) Annotation manual for the Penn Historical Corpora and the PCEEC (Release 2). Department of Linguistics, University of Pennsylvania.

田野村忠温（2012）『講座 IT と日本語 4：Ruby によるテキストデータ処理』東京：明治書院.

山本和彦（2000）『リスト遊び：Emacs で学ぶ Lisp の世界』東京：アスキー.

吉本啓（2016）「統語・意味解析情報付き日本語学コーパスの構築に向けて：アノテーション方式とコーパスの特色」『日本言語学会第 153 回大会予稿集』434–439.

吉本啓・中村裕昭（2016）『現代意味論入門』東京：くろしお出版.

文献ガイド

本文でも触れた，有用な参考書の特徴を簡単にまとめておく。

広瀬（1999）：Emacs Lisp を学んでみたい場合，まず最初にこれを読むとよい。長いこと入手しづらかったが，最近改訂版が出たので，大きな書店のプログラミング言語の棚には置いてあることがある。

山本（2000）：再帰関数の書き方の説明が詳しく分かりやすい。コンパクトだが，きちんと理解すると応用が利く内容が厳選されているため，丁寧に読む価値がある。

るびきち（2011）：この本はとにかく情報量が多いので，実際にプログラミングを始めてからレファレンスとして手元に置いておくのがよい。機能からコマンド名を調べることができる「逆引き目次」が特に便利。

Graham（1997）：Lisp はシンタクスが単純な言語である。プログラムとプログラムが扱うデータの構造が同じであるという言語設計によりメタプログラミング（コードを生成するコードを書くこと）が容易になる。

この本は，このメタプログラミングの威力を明快に解説している点で類例を見ない。
青木（2006）：関数型言語 Haskell の入門書。Haskell の本というと，抽象的な数学の概念を例にとっての説明が延々と続くことがあり，「ふつうの」プログラマーには近寄りがたいイメージを与えるが，この本は，あえて文字列操作（簡単なテキスト処理）を題材にとり，関数型プログラミングの利点，楽しさを分かりやすく説明している点がユニークで貴重。

Development of a Syntactic Annotation Tool for Parsed Corpora

KUBOTA Yusuke

University of Tsukuba / Project Collaborator, NINJAL

Abstract
This paper describes an extension of the Emacs editor for the annotation of syntactic structures in parsed corpora: “Emacs Keyaki Mode.” Keyaki Mode was developed for the purpose of aiding manual correction of syntactic annotation in the construction of the NINJAL Parsed Corpus of Modern Japanese. In the course of developing this software, we learned that the extensibility of Emacs via Emacs Lisp (which is a full-fledged programming language rather than an impoverished macro language for editor customization) is very useful and makes Emacs a potentially attractive environment for developing text annotation tools in general. At the same time, we encountered several challenges mainly due to the fact that the default interface of Emacs is somewhat idiosyncratic and unintuitive from a modern perspective. After explaining the main features of Keyaki Mode and sketching its implementation, the paper discusses potential advantages and pitfalls when Emacs is viewed as a platform for annotation tool development.

Key words: treebank, annotation, corpus, Emacs, user interface